TERRAFORM:

Terraform is an IAC tool.

It helps in provisioning the servers. (It also provides some basic configuration features for servers.)

Best practice is to use terraform only for provisioning servers.

Installation of Software and configuration is left to Software configuration management tools like Chef, Ansible and Puppet.

Installing Terraform:

1. Download the zip archive in https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli
2. Unzip the archive.
3. Move the file **mv ~/Downloads/terraform /usr/local/bin/**

Creating resources in AWS Using Terraform:

1. Creating a sample terraform file.
2. All terraform files end with **tf extension (.tf)**

main.tf

Here is where we write all the configurations.

Provider is responsible for understanding API interactions and exposing resources.

To talk to any of the cloud form - provider tag is used.

#We must install the provider individually with respect to cloud provider we are using.

```
provider "aws" {
   region = "us-east-1"
}
```

CMD Command:

1. **cd /Users/vinoth/terraform/01-terraform-basics#initializing backend**
2. **terraform init**

Creating an IAM user in AWS using root credentails:

SERVICE -> IAM -> Access Management -> Users -> Create User

UserName : command_lineXXXXXXX_user
AccessKey: AKIAT7XsXXHxIG
SecretAccessKey: /tP/XV+csnto/DUXiwwP+glmJm

Trying to set Environment variables in CMD: **export AWS_ACCESS_KEY_ID=**AKIAT7XAPXXHxIG
 export AWS_SECRET_ACCESS_KEY=tP/XV+csnto/DUXiwwP+glmJm

Creating AWS S3 Bucket using Terraform:

#creating manually in aws console
Services -> s3 -> Create Bucket -> BucketName(Unique across globe)

S3 - Simple Storage Cloud (Used to store files, backup.)

```
provider "aws" {
   region = "us-east-1"
}
```

```
#plan and execute
# Whenever we need to create something we have to include in resource tag.
#resource = providername_resouretype and internal terraform name
resource “aws_s3_bucket” “my_s3_bucket” {
#name of the bucket in aws cloud
Bucket =“my-s3-bucket-vinothkumar-6”
}
```

In terraform two step execution approach is used.

1. **Plan**
2. **Execute**

Command :

```
#- will give the overall plan details.
terraform plan
#- execute command
terraform apply
```

```
#output
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

S3 bucket will be created.

After successful bucket creation **terraform.tfstate file will be generated it will have json objects.**

1. Even if we execute the **terraform apply** without doing any changes to the file, It will just **refresh the state** and won't perform any actions.

```
#output
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Terraform recognizes nothing as changed with help of In **main.tf** terraform scripts, we specify the desired state.(we need a s3 bucket, 5 VM's)

Terraform States

#desired -KNOWN - ACTUAL

1. Terraform states are needed to track the dependencies between different resources.
2. Terraform.tfstate acts are cache to retrieve the information soon.

1. **Desired State** - Specifying what needs to be created like a S3 bucket, 5 Virtual Machines, etc.. in the cloud

2. **Known State** - Is the result of previous execution. (Known status of previous execution and resource creations. Like the file **terraform.tfstate**)

3. **Actual State** - State of Bucket in AWS.(Changes made on the bucket)

Whenever we do **terraform apply** if looks for information in **terraform.tfstate**

Whatever resources are configured, it will verify in the AWS if any information Is changed in AWS. It is called refreshing state…

And it realizes that the desired state is same as Actual state.

#changing the main.tf file renaming the bucket from my-s3-bucket-vinothkumar-6 to my-s3-bucket-vinothkumar-7.

2. If we execute the **terraform apply** with any changes to the file, It will just **refresh the state** and know the changes.

If we are creating to **rename a bucket**, it will ask below for permission to **delete existing bucket and create a new one.**

#output

Plan: 1 to add, 0 to change, 1 to destroy.

Do you want to perform these actions? Yes

Apply complete! Resources: 1 added, 0 changed, 1 destroyed.

#changing the main.tf file. Enabling the versioning for of the bucket.

3. If we execute the main.tf file using terraform apply after including the versioning of the bucket.

```
resource "aws_s3_bucket" "my_s3_bucket"{
   bucket ="my-s3-bucket-vinothkumar-7"
   versioning {
      enabled = true
   }
}
```

Now, **desired state = versioning of bucket, Known state is terraform.tfstate of previous execution, and the actual state is whatever present in AWS.**

terraform apply

```
#output
versioning {
          ~ enabled    = false -> true
            # (1 unchanged attribute hidden)
        }

        # (2 unchanged blocks hidden)
    }
```

Plan: 0 to add, 1 to change, 0 to destroy.
Do you want to perform these actions? Yes
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

Terraform is Declarative.

#query about current state
terraform console ->
#providername_resouretype.internalTerraformName
> aws_s3_bucket.my_s3_bucket

All Bucket information can be found.
#to check if bucket versioning is enabled or out.
>aws_s3_bucket.my_s3_bucket.versioning[0].enabled
true

We can include the property in **main.tf**

```
output "my_s3_bucket_complete_details" {
        Value = aws_s3_bucket.my_s3_bucket.versioning[0].enabled
}
```

Command to execute output and see:

#refresh=false is enabled to avoid the comparison between actual state(s3 in AWS)
terraform apply -refresh=false

All bucket related information will be showed.

Creating AWS IAM User using Terraform:

#Creating an IAM user in AWS console using root credentials:
SERVICE -> IAM -> Access Management -> Users -> Create User

Changes in main.tf file to create a IAM user using terraform:

```
resource "aws_iam_user" "my_iam_user" {
   name = "my_iam_user_vino"
}
```

#Command to save the terraform plan to a file
terraform plan -out iam.tfplan

#trying to execute the plan saved in cmd to create the user.
terraform apply iam.tfplan

#Output:

```
aws_iam_user.my_iam_user: Creating...
aws_iam_user.my_iam_user: Creation complete after 2s
[id=my_iam_user_vino]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Outputting the i’am details:output "my_iam_user_details" {

```
   value = aws_iam_user.my_iam_user
}
```

#command to check output of ram user details
terraform apply refresh=false

```
Outputs:
my_iam_user_details = {
  "arn" =
"arn:aws:iam::272763436187:user/my_iam_user_vino"
  "force_destroy" = false
  "id" = "my_iam_user_vino"
  "name" = "my_iam_user_vino"
  "path" = "/"
  "permissions_boundary" = ""
  "tags" = tomap(null) /* of string */
  "tags_all" = tomap({})
  "unique_id" = "AIDAT7APXXSNQOFJUU3J4"
}
```

The **arn** (amazon resource notation) is the unique identification of any AWS resources.

Updating IAM username in terraform main.tf

We can directly change the name in resource it will delete the user and create a new user with new name.

```
resource "aws_iam_user" "my_iam_user" {
   name = "my_iam_user_vinoth"
}
```

```
#command
terraform plan -out iam.tfplan
```

Plan: 0 to add, 1 to change, 0 to destroy.

```
Changes to Outputs:
  ~ my_iam_user_details          = {
        id                  = "my_iam_user_vino"
      ~ name                 = "my_iam_user_vino" -> "my_iam_user_vinoth"
      ~ tags                = null -> {}
```

```
        # (6 unchanged attributes hidden)
    }
```

```
#command
terraform apply iam.tfplan
```

```
#output
aws_iam_user.my_iam_user: Modifying... [id=my_iam_user_vino]
aws_iam_user.my_iam_user: Modifications complete after 1s
[id=my_iam_user_vinoth]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Instead of doing the entire file check and update, just to update the user alone we could do below command.

```
#command
terraform apply -target=aws_iam_user.my_iam_user
```

T**erraform intelligently** understands that S3 bucket once created cannot be renamed so it deletes it and creates a new one with a given new name, wheres Iam username can be renamed so it does not delete the user, it renames it.

Terraform States files:

Whenever **terraform apply** is performed,

1. **terraform.tfstate** - state after successfull execution of command will be saved in **terraform.tfstate** file
2. **terraform.tfstate.backup** - state before execution of command will be saved in **terraform.tfstate.backup** file

If there is no known state I.e(**terraform.tfstate** file), it cannot compare known state with the actual state.

If we try to run a plan, it will try to **create a user and s3 bucket as per main.tf file.**

Let’s understand the reason…

Terraform works mainly on the **terraform name.**

Terraform matches the **terraform name** from **main.tf** resources with the name from the **terraform.tfstate** file. And it takes the **unique_id** from the **tfstate** and tries to match it with **AWS cloud.**

If **terraform.tfstate** is not present terraform will have no idea, so it will try to create new resources.

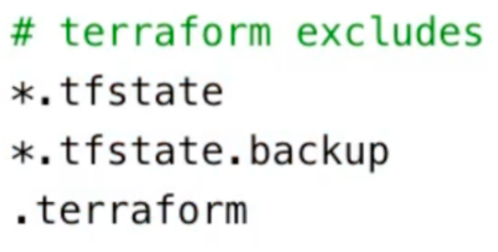
So known state(**terraform.tfstate**) is very informant.

If in a project if multiple people are working, we must share them the known state.

Terraform state is an unencrypted file containing sensitive information committing it in git or version control repo is un advisable.

We can use something called a **remote backend** in cloud like **S3**

#excluding the files for committing in git



```
# terraform excludes
*.tfstate
*.tfstate.backup
.terraform
```

Creating all the resources in a single **main.tf** file will make it complex.

The file can be any name not just **main.tf it can be any name if tf as extension.**

All tf extension files are concatenated and executed at a single time.

Destroy All the resources created with Terraform.

1. Resources created is IAM user and S3 bucket.

```
#command  to destroy all the resources
terraform destory
```

```
vinoth$ terraform destroy
```

Output:

```
aws_iam_user.my_iam_user: Refreshing state... [id=my_iam_user_vinoth]
aws_s3_bucket.my_s3_bucket: Refreshing state... [id=my-s3-bucket-vinothkumar-7]
```

terraform destroy refreshes the state, looks the current state, identifies the resources, and deletes them.

```
Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

aws_iam_user.my_iam_user: Destroying... [id=my_iam_user_vinoth]
aws_s3_bucket.my_s3_bucket: Destroying... [id=my-s3-bucket-vinothkumar-7]
```

```
aws_iam_user.my_iam_user: Destruction complete after 1s
aws_s3_bucket.my_s3_bucket: Destruction complete after 1s

Destroy complete! Resources: 2 destroyed.
```

Terraform to Create multiple IAM Users using Terraform:

```
# mentioning the count and the index of count. This is supported by HCL.

resource "aws_iam_user" "my_iam_user" {
   count = 2
   name = "my_iam_user_vinoth_${count.index}"
}


#initializing the new project to downloaded necessary providers.
terraform init

#save the plan of iam policy
terraform plan -out iam.tfplan

#execute the plan
terraform apply iam.tfplan

#output
aws_iam_user.my_iam_user[1]: Creating...
aws_iam_user.my_iam_user[0]: Creating...
aws_iam_user.my_iam_user[0]: Creation complete after 1s
[id=my_iam_user_vinoth_0]
aws_iam_user.my_iam_user[1]: Creation complete after 1s
[id=my_iam_user_vinoth_1]


#command to validate the terraform file
terraform validate
```

```
#command to format the terraform file
terrraform fmt
```

```
#command to show the current state of resources
terraform show
```

Terraform helps more in Error validation. If there are scripts with errors it handles them accordingly.

Variables in Terraform:

Variables are important because they make the terraform scripts dynamic.

```
variable “iam_user_name_prefix” {
type = string
default =“my_iam_user”
}

resource "aws_iam_user" "my_iam_user" {
   count = 3
   name = “${var.iam_user_vinoth_prefix}_${count.index}"
}
```

```
#command
terraform apply -refresh=false
```

```
#output
 Enter a value: yes

aws_iam_user.my_iam_user[2]: Creating...
aws_iam_user.my_iam_user[2]: Creation complete after 2s
[id=my_iam_user_vinoth_2]
```

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

1. Variable type by default its **any**. Unless it's explicitly mentioned.
#any, number, bool, list, map, set, object, tuple.

2. If default value is not given for any variable, terraform will ask for a value during **validate**.

3. Variable value can be **overridden** from the **default value** to **env variable** value.

export TF_VAR_iam_user_name_prefix=FROM_ENV_VARIABLE_IAM_PREFIX

Note: terraform.tfvars name is default for variable file if needed it can be overridden by **terraform apply -var-file="some-name.tfvars"**

4. Another way of overriding the variable value is by creating a file **terraform.tfvars** and provide the value from the file as below.
iam_user_name_prefix="VALUE_FROM_TERRAFORM_VARS"

output:

```
 name         = "my_iam_user_vinoth_0" ->
"VALUE_FROM_TERRAFORM_VARS_0"
```

Plan: 0 to add, 1 to change, 0 to destroy.

5. Another way of overriding the variable value is by giving value from **CMD line**
terraform plan -refresh=false -var="iam_user_name_prefix=VALUE_FROM_COMMAND_LINE"

Priority for variable value:

Value from CMD > terraform.tfvars > FROM_ENV_VARIABLES > Default value.

List and Sets Variables:

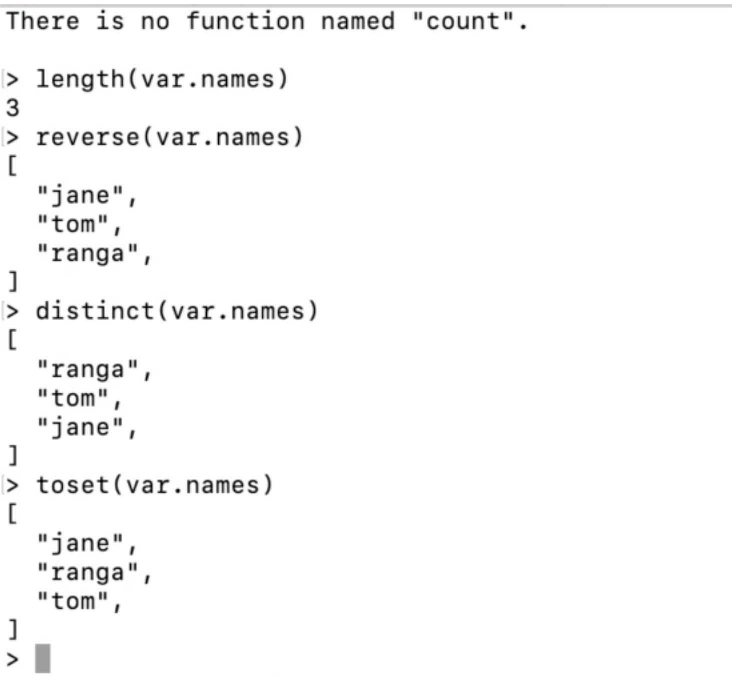
1. Creating list of usernames

```
variable "names" {
  default = ["ranga","tom","jerry"]
}

resource "aws_iam_user" "my_iam_user" {
  count = length(var.names)
  name  = var.names[count.index]
}
```

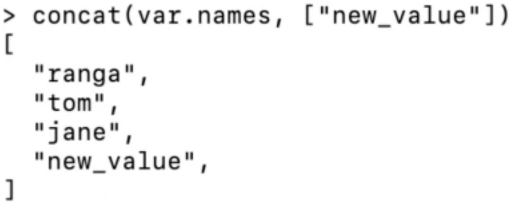
#output

```
Plan: 3 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Enter a value: yes

aws_iam_user.my_iam_user[1]: Creating...
aws_iam_user.my_iam_user[2]: Creating...
aws_iam_user.my_iam_user[0]: Creating...
aws_iam_user.my_iam_user[0]: Creation complete after 1s [id=ranga]
aws_iam_user.my_iam_user[2]: Creation complete after 1s [id=jerry]
aws_iam_user.my_iam_user[1]: Creation complete after 1s [id=tom]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

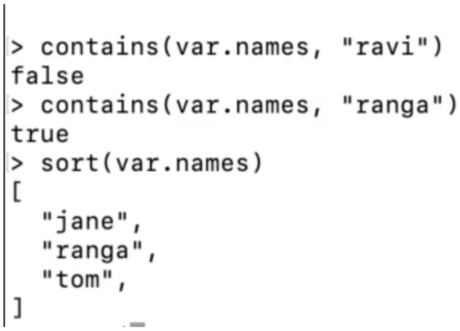
```
terraform console
```



```
There is no function named "count".

> length(var.names)
3
> reverse(var.names)
[
  "jane",
  "tom",
  "ranga",
]
> distinct(var.names)
[
  "ranga",
  "tom",
  "jane",
]
> toset(var.names)
[
  "jane",
  "ranga",
  "tom",
]
>
```



```
> concat(var.names, ["new_value"])
[
  "ranga",
  "tom",
  "jane",
  "new_value",
]
```



```
> contains(var.names, "ravi")
false
> contains(var.names, "ranga")
true
> sort(var.names)
[
  "jane",
  "ranga",
  "tom",
]
```

```
variable "names" {
  default = ["ranga", "tom", "jerry"]
}

resource "aws_iam_user" "my_iam_user" {
  count = length(var.names)
  name  = var.names[count.index]
}
```

Whenever we change the order in list in the list of names we added above,

```
variable "names" {
  default = [“sata”,”ranga", "tom", "jerry"]
}
```

Now as we added Sata in the beginning of the list

When we apply the changes

#output
Plan: 1 to add, 3 to change, 0 to destroy.

Because, we are trying to change the position in the list , all the users has to be shifted.

Whenever we add the variable with **list or count**, **terraform** stores the variables as **list of elements**. Each of them are **indexed** as list. List index compares with each element trying to changes all.

We can overcome this by using foreach

```
variable "names" {
  default = ["sata","ranga", "tom", "jerry"]
}

resource "aws_iam_user" "my_iam_user" {
  #count = length(var.names)
  #name  = var.names[count.index]
  for_each = toset(var.names)
  name = each.value
}
```

Variables can be iterated using **foreach** only if it’s **unique**. So we are **converting it into set** and iterating the values.

Now even if we add new variables, it won’t be a problem.

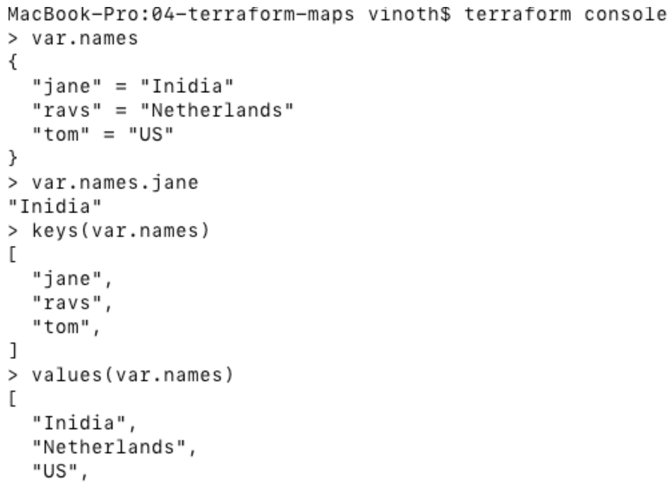
Index will the variables names.

NOTE:

1. Deletion and updating will be based on the **index in case of using count**
2. Deletion and updating will be based on the **variable name in case of using foreach.**

Variable Names with Sets:

```
variable “users” {
   default = {
      ravs:"Netherlands",
      tom:"US",
      jane:"Inidia"
   }
}

resource "aws_iam_user" "my_iam_users" {
   for_each = var.users
   name = each.key
   tags = {
      country: each.value
   }
}
```



```
MacBook-Pro:04-terraform-maps vinoth$ terraform console
> var.names
{
  "jane" = "Inidia"
  "ravs" = "Netherlands"
  "tom" = "US"
}
> var.names.jane
"Inidia"
> keys(var.names)
[
  "jane",
  "ravs",
  "tom",
]
> values(var.names)
[
  "Inidia",
  "Netherlands",
  "US",
```

Output:

```
 Enter a value: yes

aws_iam_user.my_iam_users["jane"]: Creating...
aws_iam_user.my_iam_users["tom"]: Creating...
aws_iam_user.my_iam_users["ravs"]: Creating...
aws_iam_user.my_iam_users["tom"]: Creation complete after 1s [id=tom]
aws_iam_user.my_iam_users["jane"]: Creation complete after 1s [id=jane]
aws_iam_user.my_iam_users["ravs"]: Creation complete after 1s [id=ravs]
```

Maps of maps:

Adding country

```
variable "users" {
    default = {
        ravs: { country: "Netherlands”, department: “BDA”},
        tom: { country: "US”, department: “ADS”},
        jane: { country: "India”, department: “Ds”}
    }
}

resource "aws_iam_user" "my_iam_users" {
    for_each = var.users
    name = each.key
    tags = {
        country: each.value.country
        department: each.value.department
    }
}
```

Adding another variable department

```
variable "users" {
    default = {
        ravs: { country: "Netherlands", department: "BDA"},
        tom: { country: "US", department: "BDSA"},
        jane: { country: "India", department: "BSDVDA"}
    }
}

resource "aws_iam_user" "my_iam_users" {
    for_each = var.users
    name = each.key
    tags = {
        country: each.value.country
        department: each.value.department
    }
}
```

EC2 in Amazon Console (EC2 is virtual server in AWS Cloud)

Service -> EC2 -> Launch Instance -> choose the below configurations.

1. **Choose region**
2. **Choose AMI**
3. **Choose VPC**
4. **Choose Subnet**
5. **Choose Storage**
6. **Choose security group**
7. **Tags in any**
8. **Launch Instance**

Regions:
By having multiple regions we can provide high availability and low latency for users. .

Availability Zone:Each region have multiple availability zone. AZ’s are within the regions but are physically separated data center. (For availability)

VPC:

Virtual Private Cloud is firewall for cloud. It has subnets.

Subnet:

Subnet can be **private** or **public**.

Security Group:Way to control traffic to EC2 instance. Additional traffic control other than Subnet for specific Ec2 instances.
Ingress - From where the traffic should be allowed from.
Egress - What can be done from the HTTP server.

A default security group given will have default egress for everything(to any system on the ip)

VM - Virtual Server

EC2 - Elastic Compute Cloud

US EAST (N. Virginia) us-east-1
ami-0a699202e5027c10d
t2.micro
vpc-0e1388f6633187dae

Steps:

1. Choose a region where we want to create the EC2 instance in.
   US EAST (N. Virginia) us-east-1
2. Service -> EC2 -> Launch Instance -> choose the below configurations.
3. We have to choose Amazon Machine Image(AMI) (OS and the software to be present in the server.)-> ami-0a699202e5027c10d
4. Choose the instance type (type, cpu, Memory, Storage, network performance, I-support) -> t2.micro
5. Choose the **network** and **Subnet**:I. Whenever we create a resource in data center they are already protected from a **physical firewall**. For cloud we have to create a **Virtual Private Cloud(VPC).**II. In **VPC** we can create multiple **Subnets**. - If subnet is private, only the resources within the VPC will be able to talk to the subnet and not from outside (**Databases** that shouldn't be allowed to talk to outside the from VPC we can put into **private subnets**) - If subject is public, for the resources can talk with VPC and send requests (**Webservers**, can be put into public subnets)
6. For **each region** a **VPC** is given by **default** - vpc-0e1388f6633187dae

7. Choose **storage**

8. Create **security group**. For SSH, HTTP, HTTPS and give custom ip addresses.

Creating EC2 instance Using Terraform

Service -> EC2 -> Network & Security -> Security Groups

1. Creating a security group.

Egress has to be explicitly mentioned in terraform. It is not configured by default.

1. If there is a change in name of the security group, it will delete the existing resource and create a new resource.

```
provider "aws" {
  region = "us-east-1"
}

# Http server -> 80 TCP, SSH 22 TCP, CIDR(Used to specify range of ip addresses.) ["0.0.0.0/0"]

resource "aws_security_group" "http_server_sg" {
  name   = "http_server_sg"
  vpc_id = "vpc-0e1388f6633187dae"

  # Ingress for HTTP
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Ingress for SSH
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Egress for all traffic
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"  # All protocols
    cidr_blocks = ["0.0.0.0/0"]
  }
#tags are used to tie the resouce to a specific environment which helps in identification
  tags = {
    name = "http_server_sg"
  }
}
```

Output:

```
Plan: 1 to add, 0 to change, 0 to destroy.
    terraform apply "ec2plan.tfplan"
MacBook-Pro:05-ec2-instances vinoth$ terraform apply ec2plan.tfplan
aws_security_group.http_server_sg: Creating...
aws_security_group.http_server_sg: Creation complete after 4s [id=sg-0343ad64e36d92184]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

2. Creating a Key Pair

Service -> EC2 -> Network & Security -> Key Pair

Key pair is required to run command from SSH to EC2 instances.

1. **Create key pair**
2. **Name : default-ec2**
3. **Key pair type: RSA**

4. Key file format: .pem (For use with openSSH)
5. Created.

#commands to save the file in system to protect it

```
Chmod 777 default-ec2.pem
chmod 400 default-ec2.pem
mkdir ~/aws
MacBook-Pro:Downloads vinoth$ mkdir ~/aws/aws_keys
MacBook-Pro:Downloads vinoth$ mv default-ec2.pem ~/aws/aws_keys
```

Creating a Ec2 Instance(http Server) in Terraform:

```
resource "aws_instance" "http_server" {
  ami ="ami-0a699202e5027c10d"
  key_name="default-ec2"
  instance_type ="t2.micro"
  vpc_security_group_ids = [aws_security_group.http_server_sg.id]
  subnet_id = "subnet-0f95b394e50f22512"
}
```

#output

```
MacBook-Pro:05-ec2-instances vinoth$ terraform apply ec2plan.tfplan
aws_instance.http_server: Creating...
aws_instance.http_server: Still creating... [10s elapsed]
aws_instance.http_server: Still creating... [20s elapsed]
aws_instance.http_server: Still creating... [30s elapsed]
aws_instance.http_server: Creation complete after 35s [id=i-0fefd5f1d42fcb537]
```

Connecting to the HTTP server using already created Key pair

```
variable "aws_key_pair" {
  default = "~/aws/aws_keys/default-ec2.pem"
}


resource "aws_instance" "http_server" {
  ami                    = "ami-0a699202e5027c10d"
  key_name               = "default-ec2"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.http_server_sg.id]
  subnet_id              = "subnet-0f95b394e50f22512"


  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ec2-user"
    private_key = file(var.aws_key_pair)



  }

  provisioner "remote-exec" {
    inline = [
      #install httpd, start, copy a file
      "sudo ym install httpd -y",
      "sudo service httpd start",
      "echo Virtual server is at ${self.public_dns} | sudo tee
/var/www/html/index.html"
    ]

  }
}
```

By applying the changes done above, will not make any changes in EC2.

#output terraform apply

```
aws_security_group.http_server_sg: Refreshing state... [id=sg-0343ad64e36d92184]
aws_instance.http_server: Refreshing state... [id=i-0fefd5f1d42fcb537]
No changes. Your infrastructure matches the configuration.
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

After creating EC2 instances, any changes done after creation is **immutable**.

To make the changes effective we have to **destroy** the **ec2** and create it again.

```
#it will destroy the server first  and the security group secondly because server depends on security group.
terraform destroy
#output

Plan: 0 to add, 0 to change, 2 to destroy.
Destroy complete! Resources: 2 destroyed.
```

```
terraform apply
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

Immutability and Why Immutable Server?

While provisioning servers using IAC, If new server if needed with a current state. It recommended to create a new server with version as v2. Once v2 is up and running we can remove the old v1 server.

While using IAC, immutable servers are recommended and used.

Remove hardcoded variable values in Terraform

aws_default_vpc - is managed by aws.

```
resource "aws_default_vpc" "default" {

}
```

```
terraform apply -target=aws_default_vpc.default
#output
Plan: 1 to add, 0 to change, 0 to destroy.
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

```
#default value harcooding can be avoided by this. Now the vpc_id is dynamic
resource "aws_security_group" "http_server_sg" {
  name   = "http_server_sg"
  //vpc_id = "vpc-0e1388f6633187dae"
  vpc_id = aws_default_vpc.default.id
```

```
#removing the hard coding value of subnets using data provider

data "aws_subnet" "default_subnets" {
  vpc_id = aws_default_vpc.default.id
}

  #subnet_id = data.aws_subnet.default_subnets.id
```

```
#removing the hard coding value of AMI


data "aws_ami" "aws_linux_2_latest" {
  most_recent = true
  owners      = ["amazon"]
  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*"]
  }
}
```

```
resource "aws_instance" "http_server" {
  ami = "ami-0a699202e5027c10d"
  #ami = data.aws_ami.aws_linux_2_latest
```

```
 terraform apply -target=data.aws_ami.aws_linux_2_latest
```

Terraform Graph (Dye Graph)

It is the resource graph.

Remote Backend for Storing the Tf States - S3

1. Store all the state of all projects in S3 bucket.
2. Locking - To avoid Concurrency between uses trying to use state with help of DynamoDB for locking and isolation.

  I. Acquire a lock.
  II. Update the resources.
  III. Release the lock.

3. Encryption.

Partition the resources into two folders:

07-backend-state/backend-state -> Where all the bucket configuration gonna take place
07-backend-state/users - > All user projects and other details.

07-backend-state/backend-state/main.tf:

```
provider "aws" {
  region = "us-east-1"
}
```

```
resource "aws_s3_bucket" "getajob_backend_state" {
  bucket = "application-name-backend-state-vinoth"
  lifecycle {
    prevent_destroy = true
  }
  versioning {
    enabled = true
  }
  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}

resource "aws_dynamodb_table" "enterprise_backend_lock" {
  name         = "dev_application_locks"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

```
Terraform apply
#output

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

07-backend-state/users/main.tf:

```
variable "application_name" {
  default = "01-backend-state"
}
```

```
variable "project_name" {
  default = "users"
}

variable "environment" {
  default = "dev"
}

terraform {
  backend "s3" {
    bucket = "application-name-backend-state-vinoth"
  #  key           = "${var.application_name}-${var.project_name}-${var.environment}"
  key           = "application_name-project_name-environment"
    region        = "us-east-1"
    dynamodb_table = "dev_application_locks"
    encrypt       = true
  }
}

provider "aws" {
  region = "us-east-1"
}

resource "aws_iam_user" "my_iam_user" {
  name = "my_iam_user_vinoth"
}
```

Now the keys will be saved in s3.

Creating multiple environments using Terraform Workspace:

All details for an environment it can be stored in workspace.

To switch between workspace :

```
terraform workspace select default
```

terraform workspace
#Output
-default

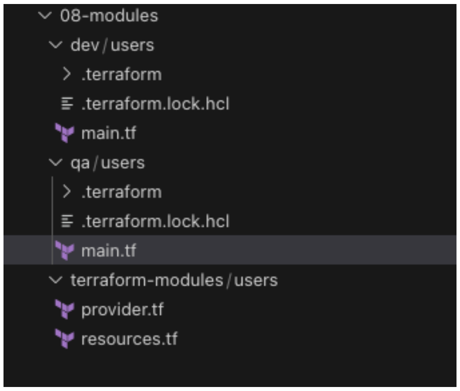
#creating a new workspace
terraform workspace new prod-env

This is create the environment in s3 bucket.

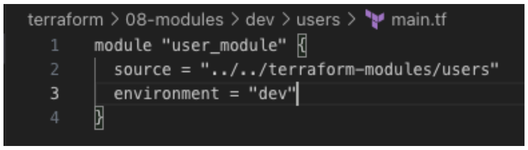
#to list all the workspace
terraform workspace list

#to show current workspace environment
terraform workspace show

Modules in Terraform:

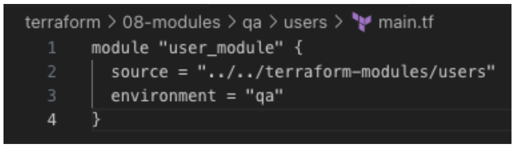


```
08-modules
  dev/users
    .terraform
    .terraform.lock.hcl
    main.tf
  qa/users
    .terraform
    .terraform.lock.hcl
    main.tf
  terraform-modules/users
    provider.tf
    resources.tf
```



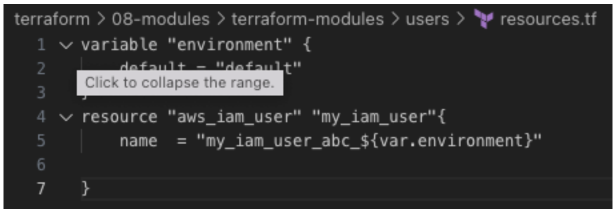
terraform > 08-modules > dev > users > main.tf

```
module "user_module" {
  source = "../../terraform-modules/users"
  environment = "dev"
}
```



terraform > 08-modules > qa > users > main.tf

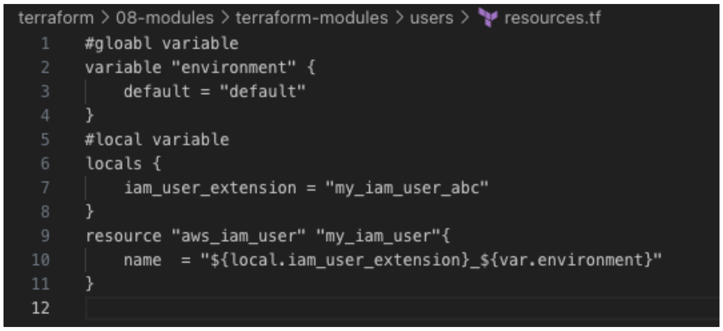
```
module "user_module" {
  source = "../../terraform-modules/users"
  environment = "qa"
}
```


terraform > 08-modules > terraform-modules > users > resources.tf
Click to collapse the range.

```
variable "environment" {
    default = "default"
}
resource "aws_iam_user" "my_iam_user"{
    name  = "my_iam_user_abc_${var.environment}"

}
```

Variables are two types:

1. Global
2. Local - variables cannot be overridden


terraform > 08-modules > terraform-modules > users > resources.tf

```
#gloabl variable
variable "environment" {
    default = "default"
}
#local variable
locals {
    iam_user_extension = "my_iam_user_abc"
}
resource "aws_iam_user" "my_iam_user"{
    name  = "${local.iam_user_extension}_${var.environment}"
}

```